An Efficient Vector/Matrix Multiply Routine using MMX™ Technology

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

CONTENTS

1.0 INTRODUCTION

Calculations on matrices of 16-bit signed integers are common in many programming problems. This application note demonstrates significant speed up of a vector dot product and a Matrix Multiply routine. It also shows, how loop unrolling can be used to optimize the performance of MMX™ technology based code.

This application note starts with the development of a vector/matrix multiply algorithm for MMX technology. Significant speedup comes from using the MMX technology inherent single instruction, multiple data (SIMD) capabilities, where arithmetic operations are executed on four operands in parallel.

The second part of this application note addresses other issues that impact program performance: redundant cache fills, poor utilization of the Pentium® processor's two execution pipelines, processor stalls waiting for not-yet-available results. These are eliminated by loop unrolling, and subsequent careful instruction scheduling to achieve optimal pairing and latency hiding.

2.0 VECTOR/MATRIX MULTIPLY ALGORITHM

The vector/matrix multiplication routine receives a Y_SIZE input vector and an X_SIZE by Y_SIZE matrix. It stores a X_SIZE resultant vector. A C code version of the routine is shown in Listing 1.

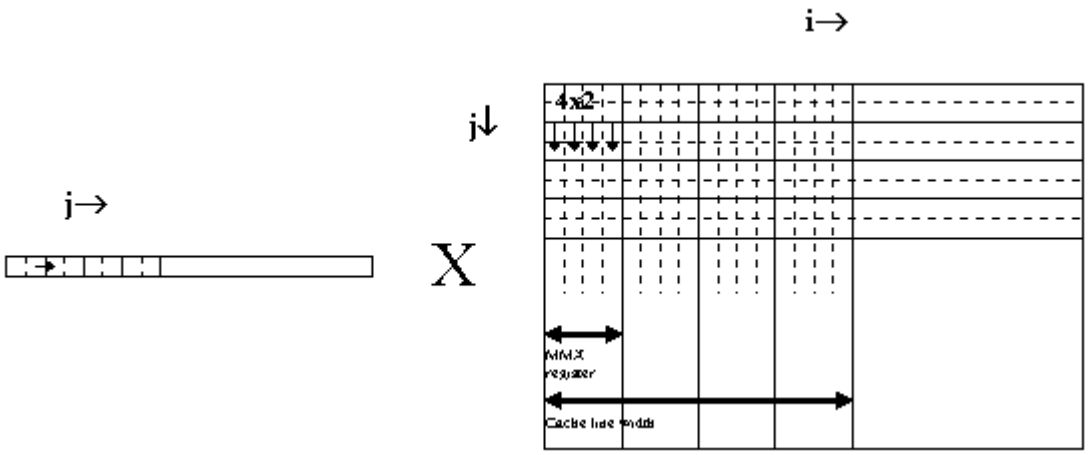
Listing 1. C code for native vector/matrix multiply routine.

```
int16 vect[Y_SIZE];
int16 matr[Y_SIZE][X_SIZE];
int16 result[X_SIZE];
int32 accum;
for (i=0; i<X_SIZE; i++) {
        accum=0;
        for (j=0; j<Y_SIZE; j++)
                accum += vect[j]*matr[j][i];
         result[i]=accum;
}
```

3.0 MMX TECHNOLOGY ALGORITHM

The native vector/matrix multiply algorithm traverses the matrix in columns. Matrices are typically arranged in row order leaving the column elements scattered in memory. Therefore, the straight forward approach - applying SIMD techniques to the inner loop - is not feasible. Instead, in an algorithm for IA media extensions, the matrix is split into elements of 4x2, and the input vector is split into elements of two (see Figure 1). The overall task can be broken down into multiplications of these fragments, as shown in C-style (augmented with vector operation constructs) in Listing 2:

Figure 1. Subdividing the input vector and the matrix for MMX technology



Listing 2. Pseudo C code for vector matrix multiply routine, modified for MMX technology.

```
int16 vect[Y_SIZE];
int16 matr[Y_SIZE][X_SIZE];
int16 result[X_SIZE];
int32 accum[4];
for (i=0; i<X_SIZE; i+=4) {
        accum = {0,0,0,0};
        for (j=0; j<Y_SIZE; j+=2)
                accum += MULT4x2(&vect[j], &matr[j][i]);
        result[i..i+3] = accum;
}
```

The revised algorithm works on four columns of the matrix in parallel, and accumulates results in a set of four accumulators. Each iteration of the outer loop yields four results. Overall, the number of iterations through the loops is reduced by a factor of 8, at the cost of more work per iteration (**MULT4x2** vs. simple multiply/add).

Listing 3 shows the MMX technology implementation of **`MULT4x2`**. The ESI register is used as a pointer to the vector; the 4x2 block is addressed through register EDX. ECX contains the number of elements per matrix line. The code shown is not optimized.

Listing 3. MMX code for `MULT4x2`.

```
movd       mm7, [esi]          ; Load two elements from input vector
punpckldq  mm7, mm7            ; Duplicate input vector: v0:v1:v0:v1
```

```
movq       mm0, [edx+0]         ; Load first line of matrix (4 elements)
movq       mm6, [edx+2*ecx]     ; Load second line of matrix (4 elements)
movq       mm1, mm0             ; Transpose matrix to column presentation
punpcklwd  mm0, mm6             ; mm0 keeps columns 0 and 1
punpckhwd  mm1, mm6             ; mm1 keeps columns 2 and 3
pmaddwd    mm0, mm7             ; multiply and add the 1st and 2nd column
pmaddwd    mm1, mm7             ; multiply and add the 3rd and 4th column
paddd      mm2, mm0             ; accumulate 32 bit results for col. 0/1
paddd      mm3, mm1             ; accumulate 32 bit results for col. 2/3
```

MULT4x2 leaves four 32 Bit results in the high and low dwords of registers mm2 and mm3. At each completion of the inner loop, these must be packed to 16 Bit and stored into the result vector, as shown in Listing 4. The code shown is not optimized.

Listing 4. Packing from 32 to 16 Bit and store into output vector

```
packssdw   mm2, mm2             ; Pack the results for columns 0 and 1 to 16 Bits
packssdw   mm3, mm3             ; Pack the results for columns 2 and 3 to 16 Bits
punpckldq  mm2, mm3             ; All four 16 Bit results in one register (mm2)
movq       [edi], mm2           ; Store four results into output vector
```

4.0 PERFORMANCE ANALYSIS

Using the code fragments from Listings 2 and 3, a vector/matrix multiply routine benefits already from the capabilities of MMX technology. For further optimization, the following issues need to be addressed:

- Pairing: For full utilization of the Pentium processor's two execution units, certain coding rules must be followed. Examples of MMX technology operations which do not pair with each other are: operations depending on each other, two multiply operations, two shift/pack/unpack operations, two operations referencing memory.

- Pipeline stalls: The Pentium processor can issue two MMX instructions every clock cycle. However, the result will not be available in the next clock cycle in all cases: Multiply operations have one cycle throughput, but three cycle latency. MMX register values must be available one cycle in advance before being written to memory. Attempting to use a result too early will stall the CPU.

- Redundant cache accesses: As cache loads are very expensive operations (11 cycles penalty best case, if the data resides in L2 cache), it is desirable to compute all data of a cache line (32 Bytes), once it has been brought into memory, rather than reloading the same data at a later point in time.

Assuming all data is in L1 cache, the code from Listing 3 takes 10 cycles for 11 instructions. The number for Listing 4 is five for four instructions. This is only half of the Pentium Processor's peak throughput, however. Both code pieces are very dense; and dependencies between the operations prevent improvement by code rearrangement.

In addition, the code shows poor cache utilization: Each cache line spans 32 bytes, or four 4x2 blocks, arranged horizontally in the large matrix. As the large matrix is traversed vertically, only 8 bytes of the cache lines are computed per iteration. The 4x2 blocks to the right of the current 4x2 block will not be computed until after completely traversing vertically through the matrix; at this point, the cache line almost certainly has been evicted and needs to be loaded again. Overall, this results in two cache loads per 4x2 block, with a (best case) penalty of 22 cycles, far exceeding the actual computation cost. If the data is not in L2 cache, the compute vs. cache load ratio becomes even worse.

5.0 MMX TECHNOLOGY ALGORITHM WITH UNROLLED LOOPS

Listing 5. Pseudo C code for vector/matrix multiply routine, four times unrolled

```
int16 vect[Y_SIZE];
int16 matr[Y_SIZE][X_SIZE];
int16 result[X_SIZE];
int32 accum[16];
for (i=0; i<X_SIZE; i+=16) {
        accum = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
        for (j=0; j<Y_SIZE; j+=2) {
                accum[0..3] += MULT4x2(&vect[j], &matr[j][i]);
                accum[4..7] += MULT4x2(&vect[j], &matr[j][i+4]);
                accum[8..11] += MULT4x2(&vect[j], &matr[j][i+8]);
                accum[12..15] += MULT4x2(&vect[j], &matr[j][i+12]);
        }
        result[i..i+15] = accum;
}
```

Listing 5 shows a version of the vector/multiply routine where the outer loop is unrolled four times. This algorithm works on 16 columns of the matrix in parallel, and accumulates results in a set of 16 accumulators. Each iteration of the outer loop yields 16 results. Unrolling the routine allows more headroom for optimization:

- An immediate benefit of the modified algorithm is perfect cache utilization. The four 4x2 blocks computed in one iteration are exactly contained in two cache lines.
- Some improvements result from reduction of loop overhead and loading of the input vector.
- Interleaving and scheduling instructions in different temporary registers achieves near perfect pairing and reduces CPU stalls. This is possible because of the data independence between the four instances of **MULT4x2.**

The four instances of **MULT4x2** in Listing 6 run in 22 cycles. The following register allocations are used:

	Instance 1	**Instance 2**	**Instance 3**	**Instance 4**
4x2 matrix	mm0, mm1	mm2, mm3	mm0, mm1	mm2, mm3
Accumulators	[eax], [eax+8]	mm4, [eax+16]	[eax+24], [eax+32]	mm5, [eax+40]

Note that the eight available MMX registers cannot hold all accumulators. Therefore, EAX is used to point to a range of temporary variables in memory. These accumulations require two instructions (paddd reg, [eax]; movq [eax], reg) rather than one (paddd reg, reg). This costs one extra cycle (as long as the memory variable is aligned to 8 Byte boundary, and the data is in cache) - a very worthwhile investment in this case.

Mm7 holds the input vector, which is the same for all four instances, and mm6 is used temporarily.

Listing 6. Instruction scheduling for four imstances of MULT$_{4x2}$

movq mm0, [edx+0] punpckldq mm7, mm7			
movq mm6, [edx+2*ecx+0] movq mm1 , mm0			
punpcklwd mm0, mm6			paddd mm3, [eax+40]
punpckhwd mm1, mm6			paddd mm5, mm2
pmaddwd mm0, mm7			movq [eax+40], mm3
pmaddwd mm1, mm7	movq mm2, [edx+8]		
	movq mm6, [edx+2*ecx+8] movq mm3 , mm2		
paddd mm0, [eax]	punpcklwd mm2, mm6		
paddd mm1, [eax+8]	punpckhwd mm3, mm6		
movq [eax], mm0	pmaddwd mm2, mm7		
movq [eax+8], mm1	pmaddwd mm3, mm7		
		movq mm0, [edx+16]	
		movq mm6, [edx+2*ecx+16] movq mm1 , mm0	
	paddd mm3, [eax+16]	punpcklwd mm0, mm6	
	paddd mm4, mm2	punpckhwd mm1, mm6	
	movq [eax+16], mm3	pmaddwd mm0, mm7	
		pmaddwd mm1, mm7	movq mm2, [edx+24]
			movq mm6, [edx+2*ecx+24] movq mm3 , mm2
		paddd mm0, [eax+24]	punpcklwd mm2, mm6
		paddd mm1, [eax+32]	punpckhwd mm3, mm6
		movq [eax+24], mm0	pmaddwd mm2, mm7
		movq [eax+32], mm1	pmaddwd mm3, mm7

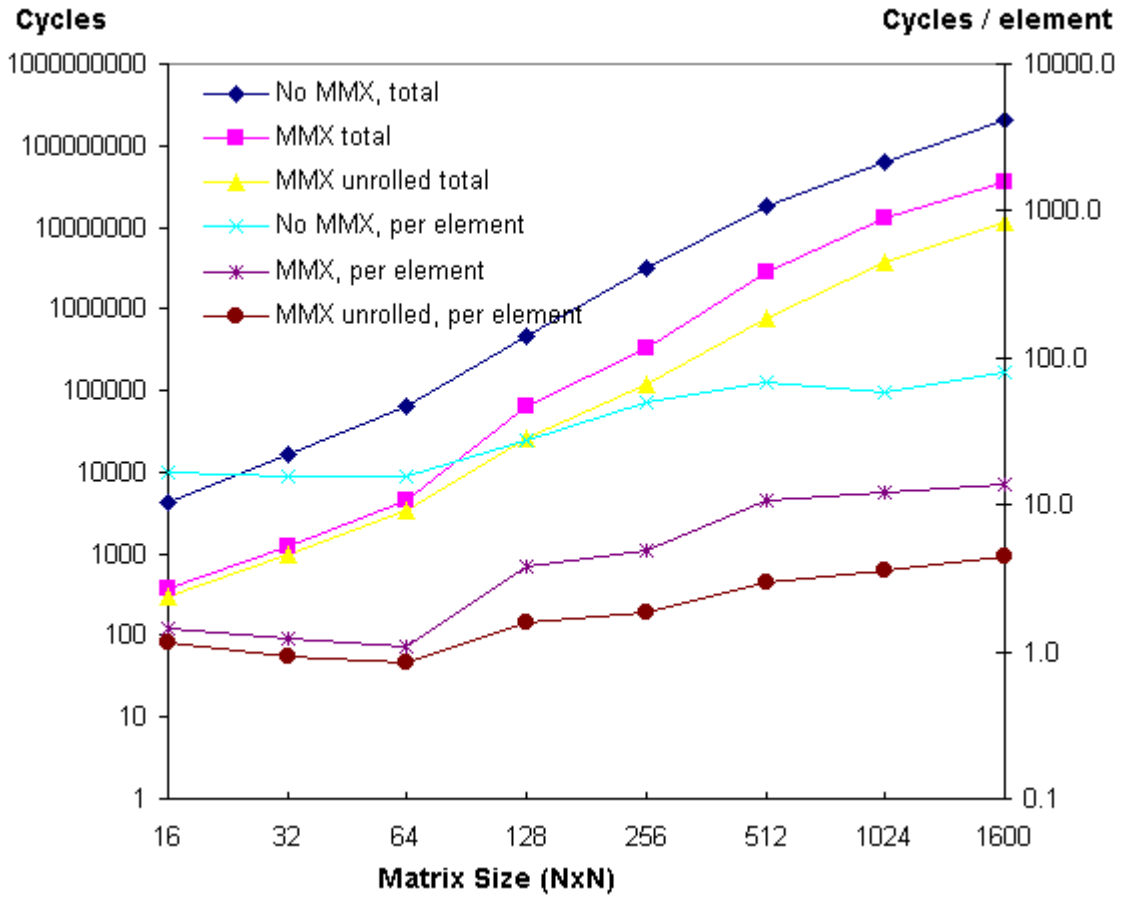
6.0 PERFORMANCE GAINS

The performance of the three multiply algorithms was measured on NxN matrices, with N ranging from 16 to 1600. All three routines have been programmed in assembly language. The results are displayed in figure 1.

On small matrices, which fit into the L1 cache, the unoptimized MMX technology based version runs more than 10 times faster than a version without MMX instructions (eg. 367 vs. 4263 cycles for 16x16 elements). The loop unrolled version gains some additional perfomance due to improved code scheduling (293 cycles).

For larger matrices, cache effects start impacting the performance of the routine. Due to the increased portion of time spent in cache loads rather than in calcualations, the performance advantage of the unoptimized MMX technology based version decreases to 5-6 (eg. 35.1 million cycles vs. 201 million cycles for 1600x1600 matrices). On the other hand, the optimized version profits from its optimal cache access pattern and runs 3-4 times faster than the unoptimized MMX technology based version (eg. 11.2 million cycles for 1600x1600 matrices). As the algorithm's performance depends heavily on the cache subsystem's performance characteristics, the numbers will vary from system to system.

Figure 1. Cycle counts for various matrix sizes



7.0 SOURCE LISTING

The code presented here assumes a even number of rows and a multiple of 16 as the number of columns. Extension to the general NxM case is straight-forward.

For optimum performance, the matrix has to be 32 byte aligned. Misaligned matrices will still deliver the correct result, but will significantly reduce performance. A misaligned input vector has minor influence on the overall performance.

For comments on the inner loop, please refer to the text on listing 6.

```
TITLE   vxmmmx
.586
.model FLAT
.data
.const
.code
COMMENT ^
void VxMmmx (
    int16 *vect,
    int16 matr[Y_SIZE][X_SIZE],
    int16 *res_asm,
    int , /* X_SIZE */
    int   /* Y_SIZE */ ) ;
^
VxMmmx PROC NEAR C USES edi esi ebx,
            vect:PTR SWORD, matr:PTR SWORD,
            res:SDWORD, x_size:PTR SWORD, y_size:PTR SWORD
        mov     edi,res
        mov     ebx,y_size
        mov     ecx,x_size
        mov     edx,matr
        sub     esp, 52         ; space for 6 accumulators plus
        lea     eax, [esp+4]    ; 4 Bytes padding (cache alignment).
        and     eax, 0fffffff8h
        pxor    mm2, mm2
        pxor    mm3, mm3
        movq    [eax], mm4
        pxor    mm4, mm4                ; clear accumulators and temp vars
        movq    [eax+8], mm4
        movq    [eax+16], mm4
do_next_16_columns:
        mov   esi,vect
        movq    [eax+24], mm4
        movq    [eax+32], mm4
        movq    [eax+40], mm4
        pxor    mm5, mm5
do_next_block:
        movd    mm7,  [esi]             ;load two elements from vector
        ; ************** four interleaved 4x2 multiplies
        movq            mm0,  [edx+0]
        punpckldq       mm7, mm7                                ;expand vector
        movq            mm6,  [edx+2*ecx+0]
        movq            mm1, mm0
        paddd           mm3,  [eax+40]
```

```
punpcklwd       mm0, mm6
paddd           mm5,  mm2
punpckhwd       mm1, mm6
movq            [eax+40],  mm3
pmaddwd mm0, mm7
movq            mm2,  [edx+8]
pmaddwd mm1, mm7
movq            mm6,  [edx+2*ecx+8]
movq            mm3, mm2
paddd           mm0,  [eax]
punpcklwd       mm2, mm6
paddd           mm1,  [eax+8]
punpckhwd       mm3, mm6
movq            [eax],  mm0
pmaddwd mm2, mm7
movq            [eax+8], mm1
pmaddwd mm3, mm7
movq            mm0,  [edx+16]
movq            mm6,  [edx+2*ecx+16]
movq            mm1, mm0
paddd           mm3,  [eax+16]
punpcklwd       mm0, mm6
paddd           mm4,  mm2
punpckhwd       mm1, mm6
movq            [eax+16], mm3
pmaddwd mm0, mm7
movq            mm2,  [edx+24]
pmaddwd mm1, mm7
movq            mm6,  [edx+2*ecx+24]
movq            mm3, mm2
paddd           mm0,  [eax+24]
punpcklwd       mm2, mm6
paddd           mm1,  [eax+32]
punpckhwd       mm3, mm6
movq            [eax+24], mm0
pmaddwd         mm2, mm7
movq            [eax+32], mm1
pmaddwd         mm3, mm7

; *********** end of four interleaved 4x2 multiplies
lea             edx,[edx+4*ecx]   ; go to next matrix line
add             esi,4
sub             ebx,2
jnz             do_next_block
paddd           mm3,  [eax+40]    ; wrapup last iteration
paddd           mm5,  mm2
; at this point, the accumulated line values can be found in:
;       0/1             [eax]           2/3         [eax+8]
;       4/5             mm4             6/7         [eax+16]
;       8/9             mm0             10/11       mm1
;       12/13           mm5             14/15       mm3
; now, pack and store 16 results. also where the V pipe can not
; be utilized, start reinitializing accumulators,
; and increment edx
movq            mm6, [eax]
packssdw        mm4, mm4
movq            mm2, [eax+16]
```

```
        packssdw        mm6, mm6
        movq            mm7, [eax+8]
        packssdw        mm2, mm2
        mov             edx, matr
        packssdw        mm7, mm7
        add             edx, 16*2
        punpckldq       mm6, mm7
        mov             matr, edx
        punpckldq       mm4, mm2
        movq            [edi], mm6
        packssdw        mm0, mm0
        pxor            mm2, mm2
        packssdw        mm1, mm1
        movq            [edi+8], mm4
        punpckldq       mm0, mm1
        movq            [eax], mm2
        packssdw        mm5, mm5
        movq            [edi+16], mm0
        packssdw        mm3, mm3
        movq            [eax+8], mm2
        punpckldq       mm5, mm3
        movq            [eax+16], mm2
        pxor            mm3, mm3
        movq            [edi+24], mm5
        pxor            mm4, mm4

        ;---------- pack and store done -----------------------

        add             edi,16 *2;sizeof(short)
        mov             ebx,y_size
        sub             x_size, 16
        jnz             do_next_16_columns
        emms
        add             esp, 52 ;release accumulator memory on stack
        ret
VxMmmx ENDP
END
```